Data-Driven Decisions and Actions in Today's Software Development

Gall, Harald ; Alexandru, Carol V ; Ciurumelea, Adelina ; Grano, Giovanni ; Laaber, Christoph ; Panichella, Sebastiano ; Proksch, Sebastian ; Schermann, Gerald ; Vassallo, Carmine ; Zhao, Jitong

Abstract: Today's software development is all about data: data about the software product itself, about the process and its different stages, about the customers and markets, about the development, the testing, the integration, the deployment, or the runtime aspects in the cloud. We use static and dynamic data of various kinds and quantities to analyze market feedback, feature impact, code quality, architectural design alternatives, or effects of performance optimizations. Development environments are no longer limited to IDEs in a desktop application or the like but span the Internet using live programming environments such as Cloud9 or large-volume repositories such as BitBucket, GitHub, GitLab, or StackOverflow. Software development has become "live" in the cloud, be it the coding, the testing, or the experimentation with different product options on the Internet. The inherent complexity puts a further burden on developers, since they need to stay alert when constantly switching between tasks in different phases. Research has been analyzing the development process, its data and stakeholders, for decades and is working on various tools that can help developers in their daily tasks to improve the quality of their work and their productivity. In this chapter, we critically reflect on the challenges faced by developers in a typical release cycle, identify inherent problems of the individual phases, and present the current state of the research that can help overcome these issues.

DOI: https://doi.org/10.1007/978-3-319-73897-0_9

Data-driven decisions and actions in today's software development

Harald Gall, Carol Alexandru, Adelina Ciurumelea, Giovanni Grano, Christoph Laaber, Sebastiano Panichella, Sebastian Proksch, Gerald Schermann, Carmine Vassallo, Jitong Zhao

Abstract Today's software development is all about data: data about the software product itself, about the process and its different stages, about the customers and markets, about the development, the testing, the integration, the deployment, or the runtime aspects in the cloud. We use static and dynamic data of various kinds and quantities to analyze market feedback, feature impact, code quality, architectural design alternatives, or effects of performance optimizations. Development environments are no longer limited to IDEs in a desktop application or the like but span the Internet using live programming environments such as Cloud9 or large-volume repositories such as BitBucket, Github, Gitlab, or StackOverflow. Software development has become "live" in the cloud, be it the coding, the testing, or the experimentation with different product options on the internet. The inherent complexity puts a a further burden on developers, since they need to stay alert when constantly switching between tasks in different phases. Research has been analyzing the development process, its data and stakeholders, for decades and is working on various tools that can help developers in their daily tasks to improve the quality of their work and their productivity. In this chapter, we critically reflect on the challenges faced by developers in a typical release cycle, identify inherent problems of the individual phases and present the current state of the research that can help overcoming these issues.

Harald Gall, Carol Alexandru, Adelina Ciurumelea, Giovanni Grano, Christoph Laaber, Sebastiano Panichella, Sebastian Proksch, Gerald Schermann, and Carmine Vassallo
Department of Informatics, University of Zurich, Switzerland, e-mail: lastname@ifi.uzh.ch

Jitong Zhao
Tongji University, Shanghai, China, e-mail: ABC

1 Introduction

When software development is portrayed, it is often shown as a rather uncoordinated activity in which some genius programmers get together to hack a program until it magically starts working. While this picture may have been arguable decades ago, professional software development is a very structured activity. It has evolved into a complex process that is heavily relying on various kinds of data about the software itself, about its execution environment, but also about feedback from its users and the market. All these data are used to fuel the development for optimizing technical aspects and user experience. Today's processes fully integrate all phases of software engineering, from requirements to coding and testing, to integration and deployment, and to operations of the software system. Boundaries of development and operations have been reduced to the extent that both activities have been integrated and optimized leading to better performing software in the field.

Today's software development is very well tracked and recorded in various forms of data: requirements data, code and bug repositories, testing data and code, configurations and continuous integration data, deployment scripts, or data in social coding platforms such as StackOverflow. Additionally, runtime data (e.g., from executions in the cloud) are collected and fed back into the development process. Software developers have to deal with a lot of infrastructure and data, be it static of dynamic data in various forms. Software experimentation provides further means to optimize feature roll-out for different user bases on a global scale. Consequently, process support has become even more essential given this mass of data and the interleaving processes.

From a business perspective, development workflows have been designed to allow the software creation to be planable by estimating efforts and monitoring progress. From a technical perspective, these workflows represent a safety net for the developers, because they both facilitate quality control and help to structure the work.

Inspired by the typical workflows of agile software development, Figure 1 illustrates one typical release cycle in a simplified workflow. Each cycle starts from a list of requirements and leads to a shipped product, with several phases in between. We have analyzed these phases and have identified several challenges that developers are facing in these individual phases. In the following, we describe these challenges, the data to be used for decision making and the actions that can be supported. For that we present opportunities to improve over the current state of the art by adapting results and tools that are proposed by recent research.

In Figure 1, the first step in a release cycle is the planning phase, in which the scope of the next release is planned (1). This step is typically not a technical problem, but driven by business decisions. It is, therefore, considered separate and out of scope for our discussion.

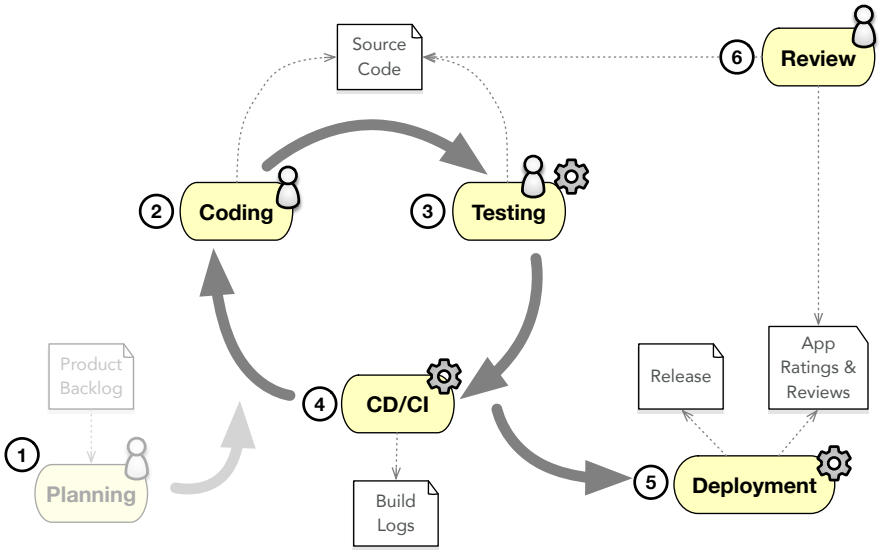

Fig. 1 Release Cycle

The core part of the release cycle is the implementation of the product itself (2). Modern software engineering has become less centered around the efficient implementation of algorithms, but more about a smart composition of existing components. Large numbers of frameworks and libraries exist that can be reused to solve a problem at hand. Reuse has several benefits like maintainability and maturity of the code base, but it requires the developer to constantly learn new application programming interfaces (API) to use these existing components effectively. Especially inexperienced developers or developers that switch projects are confronted with a huge technology stack that needs to be mastered, before they can contribute to the project This overwhelming amount of information is not only very frustrating, but also hinders productivity.

Heavily interwoven with the actual creation of source code, is the creation of test code that double checks that the implementation of the developer follows the expected behavior (3). Extensive testing is crucial to achieve a high software quality. Traditionally, developer either write small unit tests that test individual classes or resort to manual debugging to assert the expected behavior. Both approaches are suboptimal though, as important test scenarios might be missed. More recently, approaches appeared that can automatically generate test suits. Unfortunately, the quality of the generated tests is low and these suits are hard to maintain. In addition, some components are not easy to test automatically (e.g., user interfaces) and new ways of testing are required.

The most natural artifact that is created in the release cycle is source code. Both the implementation phase (2) and the testing phase (3) create source code as the primary artifact. One of the most important means for quality control in a project is a review of this source code before it can be considered finished. On top of this, source code is a way to communicate with other developers, and developers are required to understand source code on a daily basis. The high amount of source code is a big challenge for developers as it is often hard to find the important bits and pieces.

An important part of a modern release cycle is the existence of a build server that performs all integration tasks on every commit (4). Such a continuous integration/continuous deployment systems asserts that the current code base can be used on different system, e.g., it would detect, if the developers forgot to add files to a commit or broke a configuration file. In addition to asserting the deployability of the product, the build server often performs several actions to enforce several quality attributes (e.g., test coverage, code style, code complexity). When build servers fail, they typically maintain a build log that reports about the individual steps of the build. Unfortunately, the cause of the build error is often not obvious. Developers need to understand the errors reported in a build to fix them in source code or through adapting the build configuration. The length of a build log and cryptic error messages take time to process and present a big challenge for developers.

After completing several iterations in this cycle, the updated product will eventually be deployed (5). This can happen in multiple ways, most commonly, a release of the product is created and published as an update to its users. For long-running projects, the number of existing releases is large and projects that exercise continuous delivery principles excessively might create multiple releases per day. Analyzing the historic development of these releases is not only interesting for research, but also for project managers that want to study telemetry data of their product. By monitoring trends about, for example, performance metrics, code complexity, or the number of dependencies, it is possible to identify degradations early and to make educated and data-driven decisions about technical details of the product.

A cross cutting concern that is valid across all phases is the review of changes (6). We already motivated the need for summarization techniques to support efficient reviews of source code reviews, but other use cases exist. In case of a deployment to app stores, any release can also be rated by users, which creates a valuable source of feedback for the developers. The vast numbers of ratings and reviews can be leveraged to better understand the requirements and sentiments of the target users. Rating and reviews can be used to decide about the direction of future development. However, to make the amount of data digestible by a human, it is required to summarize it to the most valuable information first.

In the following, we will dedicate one section to each phase of the release cycle. We describe current tools and approaches that add value to the process, describe the data-driven aspects, the current boundaries of feasibility, and

sketch the next big idea that is about to impact the corresponding phase. While this chapter has a survey character, we will restrict the referenced works to existing approaches and to promising new results that could be put to action soon. We do, however, omit some visionary solutions for which the adoption can not be expected in the near future.

2 Recommendation

Programmers extensively leverage Application Programming Interfaces (APIs) to reuse code and unify programming experience when writing code. However, it is still a tough task for programmers to choose and utilize APIs from numerous libraries and frameworks. Even the most experienced programmers may encounter unfamiliar APIs and spend lots of time to learn how to use it. Furthermore, various barriers [67, 105] cause APIs to be hard to learn, such as insufficient examples, and ambiguous documentation. As a result, it is a critical job for assisting programmers to learn APIs effectively and efficiently with less effort.

2.1 Code Example Recommendation Systems

To help programmers alleviate burden and better facilitate the usage of APIs, developers can be supported through code example recommendation systems [98]. Over the last decade, several of such systems have been proposed [101, 107, 94]. These systems propose programming information, such as which methods are most likely to call, and how to invoke these methods. Traditional code example recommendation systems generally provide suggestions based on API usage data. Existing contributions can be organized in the categories according to the purpose of the detecting techniques.

Code search engines (CSEs), such as Krugle[1] and Nerdydata[2] usually leverage text-oriented information-retrieval (IR) techniques to search in a large number of open source repositories. But they don't provide mechanisms to rank the quality of the found code snippets, and usually return too massive results for programmers to choose.

API usage example recommendation systems utilize API examples or API calls to recommend example code. These systems adopt various categories of API usage patterns, various techniques for inferring and clustering API usage algorithms, such as DBSCAN [108], k-nearest neighbor[16], BIDE [128], RTM [82], or clustering algorithms like Canopy clustering [99]. The quality of the

[1] http://www.krugle.com/

[2] https://nerdydata.com/

API usage examples found by these tools is derived from the overall quality of the code repositories they utilize and the selected mining algorithms.

Other tools use semantic analysis approach to explore API usage obstacles through analyzing programmers questions in Q&A website. Example Overflow [129] uses keyword search based on the Apache Lucene [75] library, which internally uses the term frequency-inverse document frequency (tf-idf) weight [127]. Using Q&A website as cod repositories, systems would not be able to critically evaluate various snippets, and bugs may crop up for the examples are not properly tested.

2.2 Naturalness of Software

The implementation of recommender systems, that aid developers in writing and maintaining code, has often employed machine learning and data mining approaches. The availability of a large and growing body of successful open source projects and a recent hypothesis *"the naturalness of software"* has opened the possibility of applying natural language processing techniques to source code. The hypothesis states that software, as a form of human communication has statistical properties similar to the ones specific to natural language and that these can be exploited to build and improve software engineering tools [4].

Code suggestion is one of the most popular recommender system and most used feature of any modern IDE, it is typically implemented using manually defined syntactic and semantic completion rules derived from the programming language specification. Hindle et al. [59] observed that code corpora present a high degree of repetitiveness and they were able to exploit this property using a simple n-gram language model to enhance the code suggestion capabilities of Eclipse. Allamanis et al. [7] take advantage of the available open source code online and learn a language model using a corpus 100 times larger than the previous work, and improve their results showing that language models learned over source code, just like natural language, benefit significantly from more data. Tu et al. [120] analyze the limitation of previous models in capturing local regularities that are highly typical for human written programs and build a cache language model, that further improves the code suggestion accuracy of previous work.

In [5] the authors tackle an interesting problem using statistical language models of source code: that of learning coding conventions from a codebase. Adhering to coding conventions is an important practice of any successful and high quality software project, as it strongly impacts readability and maintainability and it is often enforced by developers, but it currently lacks support in modern IDEs. Allamanis et al. [5] learn the coding style conventions specific to a software project through simple n-gram based language models, that are subsequently used to detect style violations of identifier naming and format-

ting and suggest improvements. One of the limits of this approach, is that it can only suggest names that appear in the training set of the language models. While this is an adequate solution for local variable names, suggesting method and class names requires a more sophisticated approach. In [6], the authors experiment with a neural log-bilinear language model that is able to recommend neologism, names that do not appear in the training corpus, with promising results.

Natural language processing techniques applied to source code are extremely versatile: researchers have exploited them to evaluate code contributions to open source projects and analyse whether they are likely to be accepted or not [58], improve reporting of compiler error messages [19], help developers find buggy code that is flagged as unnatural by language models [104], etc. Nevertheless, these techniques come with their own set of challenges. Natural language and source code have different characteristics which have to be taken into account when re-using approaches built and evaluated primarily for spoken and written language. A second problem is that a basic principle of software engineering, reusability, creates a data sparsity problem: it is rare to find multiple implementations of the same task in code, while it is quite common to find many news articles written about the same topic. While programming, developers often define new terms and compose them in novel ways, current NLP methods for natural language texts have been developed expecting that this is unlikely to occur. Another important issue is the evaluation of models trained on source code, there is a need of metrics adapted to source code and of existing benchmarks for researchers to compare their results. In spite of the existing limitations, there is a wide potential to apply natural language processing methods in a wide range of areas in software engineering and support developers in writing and maintaining code.

2.3 Evaluation

Independently of the technology that is used to build a recommendation system, be it a traditional recommender learned from examples or a recommender that is build on top of a language model, it will always remain a great challenge to evaluate the value of such a system. Traditionally, researchers have built benchmarks from the vast amounts of source code found in public repositories like GitHub. They would use the source code to learn models and validate the models on other examples. This approach has one significant drawback though, these benchmarks need to be considered as artificial. Previous work has shown that the history recorded in a repository is not representative for actual development [86], because it is incomplete. They find that a representative picture requires more fine-grained evolution data.

To close this mismatch, researchers have started to collect interaction data of developers directly in the IDE. The tools DFlow [78], FeedBaG [8, 100],

or Epicea [37] are three examples of such systems that track developers during their day to day activities. The resulting datasets [96] present a unique opportunity to learn about patterns in developer behavior and to identify chances to improve their productivity in reoccurring tasks.

Examples of how to use such information are presented in several studies on the typical time-budget of developers [9], frequently used commands in the IDE [84], or to find smells in interaction sequences [31]. As some of these trackers also capture source code changes, it is possible to use the interaction data as a ground truth for the evaluation of recommendation systems in software engineering. Prior work has shown that these realistic evaluations report different quality metrics for recommenders, when the evolving nature of source code is not reflected in the evaluation setup [97].

3 Testing

It is widely recognized that software testing is a crucial task in any successful software development process. Indeed, the overall testing cost has been estimated at being about half of the total development cost [12]. The definition of software testing involves several different kinds of activities and processes. In fact, various types of testing need to be performed in order to achieve different objectives and assess the qualities and the reliability of a software system. There are two main categories in software testing. On one side *functional testing* assesses whether software behaves as intended. This category contains unit, integration, and user-acceptance testing. On the other side *non-functional testing* is concerned with program attributes like performance, security, or reliability.

Software testing is extensively handled in research, hence we focus on two specific topics from both above mentioned categories. We start off by introducing concepts from automated unit-test generation. Afterwards, performance testing in the form of software microbenchmarking is discussed.

3.1 Automated Unit Test Case Generation

Unit test is intended as a piece of code that automatically invokes a *unit* of work in a given system, checking assumptions about its behavior. To do that, inputs that exercise such units needs to be defined. However, find those inputs and write test cases for a large system is an extremely costly, difficult and laborious task. An obvious response to this problem is to automate such process as much as possible. Since the test case generation problem can be easily expressed as an optimization one [3], a tremendous amount of research has been conducted in applying metaheuristic algorithms (especially Genetics

Algorithms (GA) [50]) to solve such problem. The Search-Based Software Testing (SBST), an entirely new line of research, is the result of such growing interest in the area [76].

The design of a search algorithm tailored to solve any optimization problem usually starts from the definition of the solution representation and the fitness function. In this context, a solution is represented by a set of test inputs. The fitness function is used to represent how *good* is a given solution for a coverage criterion. The most common one is the brach coverage criterion. A fitness function is composed mainly by two metrics, the *approach level* and the *branch distance*. The first one express how far is the actual execution path from covering the target; the latter represents how far is the input data to change the boolean value in the closer condition node to the target. Depending on how the targets are handled by the evolutionary algorithm, we can distinguish between single-target and multi-target approaches.

3.1.1 Single-Target Approaches

This class of algorithms has been the first one proposed in the literature as a search-based approach for test case generation [41, 48, 21]. A single-target strategy works as follows: (i) all the targets to hit are listed, (ii) a single-objective search algorithm is used to find a solution to each target until all the search budget is consumed or all the targets are covered, (iii) a test suite is built combining together all the generated test cases. Therefore, in such techniques every individual is a test case that evolves to cover a target.

From the ones presented in the literature, we believe that several tools are mature enough to be used in industrial applications, especially for programs written in C language. For instance, Lakhotia et al. implemented in AUSTIN, an open-source tool based on the Alternative Variable Method method able to deal with pointers [71]. Scalabrino et al. presented OCELOT, a tool that implements a technique based on the concept of linearly independent path to smartly select the targets and therefore save search budget [109]. Moreover, such tool is able to automatically generate test cases for the Check[3] framework. More recently, Kim et al. introduced CAVM, an extension of a commercial tool able also to handle dynamic data structures [66]. Despite working pretty well for procedural languages, single-target techniques might suffer of some limitations [44]. For instance, in a program under test, some branches might be more difficult to cover, or even infeasible. Thus, in this case, a single-target approach would waste a significant amount of budget. Multi-target approaches, discussed in the upcoming paragraph, have been proposed in last years to overcome such limitations.

[3] `https://github.com/libcheck/check`

3.1.2 Multi-Target Approaches

The first example of multi-target technique has been presented by Fraser and Arcuri [44]. They proposed a *whole-suite* approach where the search algorithm evolves the entire suite with the aim to cover all the branches at the same time. In order to achieve such result they define a new fitness function that sums the branch distance of all the targets into a cumulative function that express the *goodness* of the entire suite. Such approach has been implemented in EvoSuite[4], an open source tool generating JUnit test cases for java code. Following a similar idea, Panichella et al. proposed MOSA (Many-Objective Sorting Algorithm) [88]. Instead of aggregating multiple objectives into a single values, MOSA reformulates the branch coverage as a many-objective optimization problem. Indeed, in this formulation, a fitness score is a vector of m values, instead of a single aggregate score. In addition, MOSA uses an *archive* to keep track of the best test cases between the many detected by the algorithm. Evaluated on 64 Java classes of large projects, MOSA was able to generate unit test with an average coverage about 84%. Moreover, also such algorithm is built on the top of EvoSuite. Being available as Maven plugin, such a tool represents an out-of-the-box solution for practitioners that want to automate the process of test case generation.

3.1.3 Limitations and Outlook

The aforementioned approaches only automate the process of generating data able to exercise a part of a system. However, given such input data, a proper test case should be able to check whether the software is behaving as intended, preventing it from potentially incorrect behavior. Such a problem is called the *test oracle problem* [11]. Despite the huge amount of research in testing automation, such problem still remain less solved. Therefore, without test oracle automation, human effort is needed to determine the correct behavior and inhibits better overall test automation. Moreover, such tools often generate test cases that are hard to understand and difficult to maintain [106]. Despite different approaches that tried to address such a problem [**?**], there is still room for improvement.

3.2 Performance Testing

Performance testing is a form of non-functional testing that deals with the assessment of particular performance counters of a system. A system can range from a piece of software to a deployed application potentially running on

[4] `http://www.evosuite.org`

multiple computers. Hence, we differentiate between two major types of performance tests, (i) load tests, and (ii) software microbenchmarks. During load testing a production-like system is deployed to a dedicated environment, and defined load patterns are executed against that system for a period of time (usually multiple hours). Load testing can be seen as the system/integration testing equivalent for performance. Conversely, microbenchmarking focuses on small fractions of a program (e.g., a function), and evaluates over many executions how performance counters behave for that particular fraction. It is the unit-test equivalent for performance. Typical performance counters evaluated are related to time and required resources. Examples are average execution time, throughput, and CPU utilization; and memory consumption, number of allocations, and I/O operations.

In the following we focus on software microbenchmarking for software-component execution times.

3.2.1 Problems

Recent studies on OOS show that microbenchmarking is not as common and popular as unit testing [72, 119]. The decreased popularity is potentially due to multiple factors described in the following. In order to write good performance tests, an in-depth knowledge about compiler/runtime internals and statistics is required [49] . Moreover, execution should be done on an environment dedicated and set-up for reliable performance measurements, and tests need to be executed many times (system warmup, >20 measurements) to reduce non-deterministic influences. This is results in two constraints, many developers do not have such a dedicated environment but rather use their own machines or unreliable cloud resources, and test-suite execution times rise up to multiple hours or even days [63]. Further, in most programming languages there is no established standard for writing microbenchmarks, and current tools that support agile process models (i.e., CI servers) do not provide means for continuous performance assessment [72].

3.2.2 Current Solutions

In recent years software microbenchmarking has gained interest in both academia and industry. To lower the required knowledge for writing microbenchmarks, tool vendors introduced dedicated frameworks that assist in writing good performance tests. OpenJDK introduced from version 7 on the Java Microbenchmarking Harness (JMH)[5]. Newer languages such as Go[6],

[5] http://openjdk.java.net/projects/code-tools/jmh/

[6] https://golang.org/pkg/testing/

Rust[7], and Swift[8] provide microbenchmarking framework as part of their standard library. On the academic side, Bulej et al. [17] introduce the Stochastic Performance Logic (SPL) that removes the required statistical knowledge from developers for performance test result evaluation. SPL is a declarative way of specifying assertions about a software components performance. On example could be, algorithm A must be faster than algorithm B by a factor of 2. These assertions are transformed into performance tests, and their results are validated with common statistical tests (i.e., hypothesis tests).

Others explored the identification of performance introducing code changes and the reduction of performance test execution time. Jin et al. [63] first study the characteristics of performance bugs and consequently take the insights to compute efficiency rules for performance bug detection. Auxiliary to that, Heger et al. [57] introduce PRCA an approach that utilizes unit tests and the revision history of a project to find the root cause of a performance problem. Their work bisects the git revision history to find the commit and involved methods that introduced the degradation. Both previously discussed works do not continuously check, but rather check for performance problems ad-hoc. Conversely, Huang et al. [60], Alcocer et al. [1], and de Oliveira et al. [34] propose approaches that continuously check for software performance. Huang et al. and Alcocer et al. introduce static approaches to detect potentially performance-risky commits, and based on their assessment flag a commit for benchmark execution or not. Conversely de Oliveira et al. are the first to introduce a methodology that executes a subset of a performance test suite on every commit. They employ a combination of static analysis whether a benchmark is potentially able to detect a regression, and historical dynamic benchmark execution data to predict whether the performance of a benchmark is affected by a commit. Compared to the other approaches, this reduces the benchmark suite to a subset that is of interest and executes a subset on each commit.

3.2.3 Outlook

An unsolved issue so far is the utilization of unstable environments for performance test execution. The premier example of such environments are cloud resources, mostly caused by virtualization and multi-tenancy. Further work in the area of continuously assessing software performance as part of CI needs to be done. We envision a future where performance testing is as common place as unit testing is today, where each build is automatically tested for its performance characteristics, and developers receive quick feedback about these non-functional attributes of their software.

[7] https://doc.rust-lang.org/1.7.0/book/benchmark-tests.html

[8] https://github.com/apple/swift/tree/master/benchmark

4 Continuous Delivery

Continuous Delivery (CD) is an agile software development practice where code changes are released to production in short cycles (i.e., daily or even hourly). A basic CD pipeline is composed by build, deploy and test stages [124]. This practice is one of the pillars of the agile movement and is widely adopted in both open-source and industry projects by now. The regular invocations of the build-related tools (e.g., static analysis tools) and the corresponding artifacts generated in this process (e.g., build logs) open up new opportunities to better understand the development process and to build tools that support the developers early on.

4.1 Build breakage

In the CD pipeline, a build is typically triggered during the build stage (i.e., Continuous Integration), whenever a code change is pushed in a version control system (e.g., Git). It is being checked out, compiled, tested and analyzed for code quality measures. The build can potentially fail in any of these phases due to several reasons, e.g., syntax errors, failing tests, or violations of coding conventions. Such a a failing build delays the release of a new software version. Indeed, developers have to analyzed and resolved the problems causing the build failure before being able to perform a new build. In such a scenario, release engineers spend at least 1 hour per day to fix broken builds [65] and an organization loses a lot of man-hours because of many build failures occurring during a working day.

Thus, it's crucial to support developers in (i) identifying faster and better the problem causing a build failure and (ii) fixing easily those problems.

The first step to meet those two challenges has been a deep understanding of the types of build failures. We performed a large study [123] of 34,182 build failures occurred in OSS and in a large financial organization, namely ING Nederland. The purpose of this study was to compute a taxonomy of build failures and compare the frequencies of each category in a closed (i.e., ING) and an open (i.e., Travis CI[9]) source environments.

Through the analysis of the build failures in 349 Java Maven projects from Travis CI and 418 (mostly Java) Maven projects from ING we derived a Build Failures Catalog including 20 categories. We briefly describe the most important types.

Compilation is the category including builds failed during the compilation of production and test code. A compilation of a code change might not succeed because of language constructs unsupported by the build environment or due to annotations unsupported by the installed Java VM.

[9] `https://travis-ci.org`

Dependencies are another substantial source of build failures. Typical errors in this category are invalid resource configurations or failed downloads due to unavailable artifacts.

Testing failures also break the build. This category is divided in other subcategories based on the testing activity involved in the failure (e.g., unit testing, integration testing, non-functional testing).

Code Analysis enforces code changes to follow predefined code quality criteria. Thus a build might fail because of non passed quality gates.

Deployment of an artifacts resulting from an introduced code change cause other types of build failures. The deployment environment might be set up incorrectly or the new application version simply doesn't harmonize.

Given this catalog of build failures, we compute and compare the percentages of build failures of different types for both, industrial (i.e., ING) and OSS projects. Then we analyzed differences and commonalities. Except for Dependencies, we observed a quite different distribution of build failure types in the two domains under analysis.

Specifically, integration testing failures are more frequent in industry than in OSS. Instead OSS projects exhibit more unit testing failures. Those results suggest that industrial developers are more keen on performing unit tests also before the build (see "Testing" in Figure 1), and rely on build server to catch mostly integration issues. In OSS, developers tend to delegate all testing activities to the build stage.

For business-critical projects like the one used by financial organizations, proper non functional testing is crucial. Thus, a separate node is usually used to perform time consuming testing activities, e.g., stress testing and penetration testing. Nevertheless, exclusively industrial developers (at least in ING) started to rely on the build process to spot, whenever possible, non functional issues, and specifically load testing failures. It allows them to make the identification of such non functional problems faster and reduce the time and the cost for fixing them.

There are more build failures due to static analysis in industry compared to OSS. Performing a qualitative analysis of some both industrial and OSS build failure logs, we observed that most of the OSS projects run static analysis tools directly on the build server, while industry tend to perform static analysis on a different server using tool as Sonarqube [10]. This choice implies (i) data easy to monitor and query and (ii) less overloading of the build server machine.

Finally, we observed a quite low percentage of compilation failures in both domains. This result shows how it's important to compile a code change before building it. This best practice of compiling code before building it make the identification of the error behind a compilation failure faster and easier (i.e., it's more difficult to spot a compilation error when a code change is already integrated with other changes).

[10] https://www.sonarqube.org

The results of our study suggest the need for supporting developers in maitaining their CD pipeline to make it more efficient, e.g., by deciding what to do in private builds on the developer's local machine and what to delegate to build servers, or how to improve the overall detection of the issues by anticipating the execution of non functional tests. We plan to use the taxonomy we built to make the overall process of build failure understanding faster and conceive approaches able to automate the build failure resolution.

4.2 Release Confidence and Velocity

The trend towards highly automated build, test, and deployment processes enables companies delivering their software quickly and efficiently. However, the faster a company moves, the less time is available to perform precautions to minimize the risk of releasing defective changes. Consequently, there exists an inherent trade-off between the risk of lower release quality and time to market. We investigated this trade-off and derived a model [110] based on *release confidence* and the *velocity* of releases during the course of two larger empirical studies [25, 111].

Release Confidence is the amount of confidence gained on the quality gates within a company's development and release process. Those quality gates involve automated (e.g., unit, integration, performance tests) and manual tests (e.g., user acceptance tests), and code reviews.

Release Velocity is the time it takes to assess each single quality gate starting with the developer's commit of a change until the change reaches the production environment, including the time it takes to deploy the newest version.

4.2.1 Model of Release Confidence and Velocity

Our model consists of four categories (*cautious*, *balanced*, *problematic*, *madness*), arranged on a grid from both low to high velocity and confidence. The underlying idea and vision of this model is to serve as a vehicle for self-assessment (i.e., what is my company's category) and provide guidelines on how a company can transition to other categories (e.g., increase velocity while keeping confidence high).

Cautious is the category characterized by high release confidence and low velocity. Companies put a high emphasis on testing, including both a well-maintained set of automated tests to reduce the risk of human caused errors during manual testing, but also supplemental manual tests to cover areas hard to test. Code reviews are a common practice complementing the testing phases. Manual approval processes (e.g., domain specific requirements, company policies) decrease velocity, hence reduce the number of releases.

Problematic is characterized by low confidence in a company's quality gates and low velocity. Typically, this is not a category a company is placed in by choice. Insufficiently maintained test suites (automated and manual), or test suites not covering all aspects of an application, shortages in testing personnel, the absence of code reviews, and unclear roles regarding the quality assurance are characteristic for this category. Velocity is often low as a direct consequence of this, but also due to lack of automation and architectural issues.

Madness is associated with high velocity and low confidence. Release cycles are short, companies make use of early customer feedback and reduced time to market. However, quality assurance plays a minor role, but often by choice, as companies within this category decide that the benefits of sophisticated quality assurance processes are not worth the investments. Consequently, issues might be fixed fast, but the lack of quality gates lead to risky and stressful releases. This category is often appealing for companies with smaller code bases (e.g., startups) as it enables pushing new functionality fast.

Balanced is characterized by high velocity and how confidence and portrays the vision of continuous delivery and deployment [61, 22]. Companies in this category strive for a balance of having sophisticated and highly automated quality assurance processes and code reviews (for specific critical code sections) to maintain confidence on a high level, and tool support that allows releasing by the push of a button. Moreover, this category provides a proper basis for post-deployment quality assurance techniques (i.e., continuous experimentation), testing new functionality on a small fraction of the user base first [111].

4.2.2 Transitioning between Categories

The derived model serves as a starting point discussing the consequences of the categories and allows investigating research gaps not only on how we can better support companies, but rather on how to guide (i.e., transition) them to other categories.

Increasing Velocity. One topic of raising popularity is containerization, and especially Docker. Docker allows packaging an application with its dependencies into a standardized, self-contained unit that can be used throughout development and to run on any system, being it the development machine, but also the production server. Its concept of lightweight virtualization speeds up the process of bringing a change into the production system without having to deal with different hardware and software platforms and their dependencies. In a recent study [26] we investigated the Docker ecosystem on GitHub to understand its evolution and identify quality issues. One of the findings is that there is space for improvement when it comes to the size of Docker images. Many projects rely on rather heavyweight OS images as their base image,

which somehow defeats the original purpose of lightweight containerization. Therefore, research should aim for providing guidelines and tool support allowing projects to reduce their image sizes and consequently reduce memory consumption when deployed at scale.

Increasing Confidence. Recently, the field of continuous experimentation has received increased attention by both academic research and industry (e.g., Fabijan et al. [40], Kohavi et al. [69]). The ability of experimenting with new functionality on a small fraction of the user base enables companies getting early feedback from real world users while at the same time keeping the risk manageable in case that something goes wrong. Tooling to support experimentation includes our own tool called Bifrost [112], Vamp[11], and Spinnaker[12]. Bifrost and Vamp support the automated, data-driven execution of experiments defined in a domain-specific language. Spinnaker serves as an extension to a CI system allowing to define additional steps for experimentation.

5 Deployment

After completing several iterations of the development cycle introduced in the beginning of this chapter, the updated product will eventually be deployed to its users. While professional software developers are mostly concerned with the quality and state of the current and upcoming releases, it is often desirable to reflect on the long-term evolution of a software project. Managers and project coordinators may be interested in learning how different parts of a project evolve, for example to re-allocate resources and estimate future effort [77, 68, 39, 121]. Modern version control systems present a rich opportunity for understanding the history of a project, but the wealth of information contain within them needs to be managed appropriately.

Two of the main challenges in analyzing the history of a project are the computational time and resource requirements. Running a static analysis tool, for example to detect bugs or compute various software metrics, can easily take several minutes for a single release. Repeating the analysis for hundreds or even thousands of past releases quickly becomes infeasible. Research has yielded two main avenues of solving this problem: (i) scaling analyses via additional resources, such as clustered computations, and (ii) increasing the efficiency of analyses by reducing redundant computation.

A prime example of the former is BOA [38], a server framework that allows analysts to formulate and execute analyses which are executed on a Hadoop cluster. BOA supports analyzing metadata of historical commits (for example to learn more about how developers have authored code in the past).

[11] `https://vamp.io/`

[12] `https://www.spinnaker.io/`

It also supports the analysis of Java ASTs to do perform static analyses. For industrial companies, a setup such as this can be useful if there is a large volume of historical code to be analyzed on a regular basis - otherwise the resources may lay idle.

An alternative to adding more resources is to reduce redundancies during the analyses of past revisions as much as possible. This is done, for example, by LISA, a stand-alone library for running arbitrary analyses on multi-revision graphs [2]. Changes between two subsequent revisions of a software project typically concern only a fraction of the source code, while most of the source code remains identical. LISA exploits this fact by loading source code ASTs and other graphs in such a manner that each node (which might, for example, represent a Java class) is stored only once for any range of revisions where there have been no changes. While analyzing the graphs, for example to compute code metrics, computations need only be run once for these entire ranges. This reduces the average time per revision by multiple orders of magnitude compared to the naive approach of analyzing revisions individually. This approach is advantageous when resources cannot be permanently assigned for the analysis of historical data.

No matter which way historical data is computed, the goal is to obtain actionable knowledge on the health of a project. An example for how this can be achieved is Evolizer [47], a library that draws data from both version control systems as well as bug trackers and enables the joined analysis of both resources. Evolizer has, for example, been used to link commits with bug tracking data to automatically determine which parts of the source code are more bug prone, since commits to the same file referencing bugs more frequently are likely to be more fragile. Evolizer has also been used to discover which parts of the source code co-evolve and are thus logically coupled.

Current research is addressing the challenges in obtaining linked, historical data efficiently and it has shown that using this data can provide valuable and useful indicators that can allow software developers to manage the complexity inherent in modern software systems. Exploiting historical data like this allows them to make decisions more confidently and increase the effectiveness of allocated resources, improving the quality and reliability of software in the long term.

6 Summarization Techniques for Code, Change, Testing, Software Reuse and User Feedback

In the current software industry developers are involved in a fierce competition to acquire and retain users. Thus, in this competitive market understanding the factors affecting users' experience and satisfaction and how these factors are related to software quality represents a valuable benefit for devel-

opers interested to evolve their software applications [51]. Moreover, with the introduction of Continuous Development and Continuous Integration software practices, it is becoming important for software developers to speed up development activities without hindering the reliability and quality of the produced software [23]. Thus, for *"modern software companies it is nowadays, crucial to enact a software development process able (i) to dynamically react to market requirements (i.e., users requests), (ii) delivering at same time high quality and reliable software"*. To achieve this high level goal developers have to efficiently dealing with with the huge amount of heterogeneous data [95, 85] they have to their disposal, e.g., bug reports [102, 91], source code [14, 117, 118, 74, 55, 79], test cases [124, 93, 43], mailing lists [113, 91], question and answer site (Q&A) discussions [122, 126], user feedback [35, 36, 90] and other kinds of development artifacts [95, 85].

According to its original definition or concept, a *"Summarization approach"* has the general capability of automatically **extracting** or **abstracting** key content from one or more sources of information [54], thus, determining the relevant information in the source being summarized and reducing its content (see Figure 2). Specifically, the *extraction* capability consists of *"selecting original pieces from the source document and concatenating them to yield a shorter text"* while the abstraction capability is different as it *"paraphrases in more general terms what the text is about"* [54]. Both the two categories of summaries can be either *indicative*, *informative*, or *critical*:

- ***indicative summary***: it provides a direct link to the required content or relevant sources to users, so that they can read the provided information more depth.
- ***informative summary***: it has the goal to substitute the original source of information, by mainly assembling the relevant content, presenting it in a new, more concise and structured form.
- ***critical summary (or review)***: it reports or selects the main opinions or statements related to a specific discussed topic, thus, it brings the most relevant feedback, both positives and negatives, about a given subject discussed in the source document.

Given the great potential of such approaches, in recent years *"Summarization techniques"* [54, 85] have been explored by SE researchers to conceive approaches and tools that support developers to dealing with the management of such huge amount of heterogenous data, coming from different sources of information [85, 81, 95, 93].

In this section we provide an overview of the summarization techniques explored in literature for supporting developers during program comprehension, development, maintenance and testing tasks, by leveraging the above mentioned heterogenous data. A more detailed and exhaustive literature review on summarization techniques proposed in SE research is reported in recent work [85, 81].

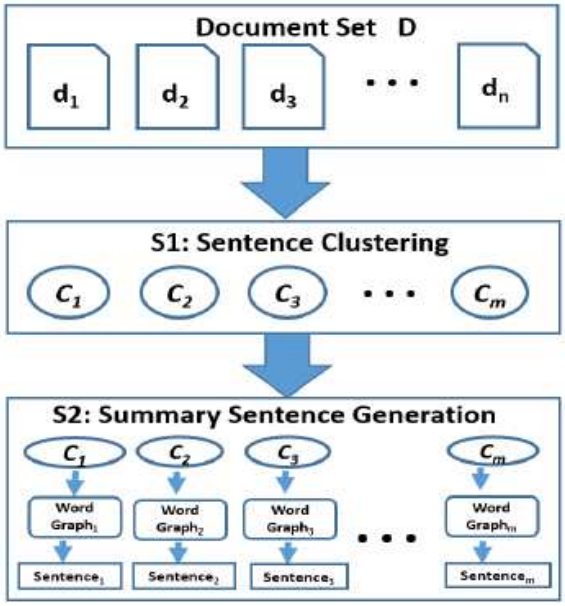

Fig. 2 High Level View of Summarization Approaches

6.1 Source Code Summarization

First attempts on the adoption of summarization techniques to SE problems where of Murphy [83] and Sridhara *et al.* [117].

The work by Murphy [83] was the first that proposed to generate summaries of source code by analysing its structural information. In particular, he proposed to use summarization techniques of such techniques for generating abstractive descriptions of its behaviour, to automatically document or re-document source code with the generated summaries. Sridhara *et al.* [117] extended such previous work by Murphy, suggesting the use of pre-defined *natural language templates*, filled with the main linguistic elements (e.g., verbs, nouns, etc.) [33, 32] composing the signature of methods, to generate the summaries.

On top of such previous work, other researchers used a similar strategy to summarize other kind of software artifacts at different level of abstraction: parameters [116], groups of statements [115], Java methods [118, 74, 55], Java classes [79], services of Java packages [56].

The main limit of such work is that they generate source code descriptions or summaries by only analyzing the static information available in the source code itself. Thus, they are not able to describe the high level behaviour and meaning of the described software artifact, something that developers often report in various communication means, such as mailing list [10, 92], issue

trackers [91, 103, 92], IRC chat log [92] and other developers communication means. For this reason, recent work proposed to generate source code documentation by mining text data from other sources of information, alternative to source code: question and answer site (Q&A) discussions [122, 126], bug reports [103, 91], e-mails [91] and forum posts [**?**].

6.2 Task Driven Software Summarization

A limit of approaches proposed for *Code Summarization* regards the way in which SE researchers evaluated their usefulness. Indeed, Binkley *et al.* [14] and Jones *et al.* [114] highlighted that the evaluation of summarization techniques for SE should not be done by using simple metrics, answering the general question *"is this a good summary?"*. Thus, they proposed the concept of *Task Driven Software Summarization* where summarization techniques for SE should be evaluated *"through the lens of a particular task"* (e.g., during bug fixing or testing tasks). Stemming from the observations made by Binkley *et al.* [14] and Jones *et al.* [114], recent work proposed approaches for automating particular software maintenance and testing tasks [81, 85], evaluating their practical usefulness in their specific utilization context.

6.2.1 Code Change Summarization

Code change summarization approaches have the goal of augmenting the context provided by differencing tools, generating natural language descriptions of changes occurred at different types of software artifacts [81, 85].

An example of such differencing tools is the *Semantic Diff* tool introduced by Jackson and Ladd [70] which detects differences between two versions of a procedure, and uses program analysis techniques to summarize the semantic differences. More recent and modern examples of differencing tools are *DeltaDoc* [18] and *Commit 2.0* [30] that (i) describe source code modifications using symbolic execution with summarization techniques and (ii) augment commit logs with a visual context of the changes, respectively. However, most of code change summarisation approaches proposed in literature [80, 28] are based on a specific, well know differencing tool called, *Change Distiller*, implemented by Fluri *et al.* [42] which extracts fine-grained source code changes based on a specialized tree differencing algorithm. Thus, Change Distiller generates a list of classified changes based on the operation type performed by the developers in the analyzed commit, i.e., *insertion*, *deletion* or *modification*. On top of such change types information it generates the corresponding, customized abstract syntax tree.

Hence, information coming from Change Distiller tools have been exploited for example by researchers to automatically generate high quality (i) commit

messages [28] and (ii) release notes [80]. Specifically, the approch for generating commit messages, called Change Scribe, conceived by Cortes *et al.* [28] takes as input two consecutive versions of a Java project then it (i) uses Change Distiller to extract the source code changes occurred between the two versions of the project (changes related to addedition, removal or modification types); (ii) detects responsibilities of methods within each Java class using the concept of method stereotypes; (iii) characterizes the change set using commit stereotypes; (iv) estimates the impact set for the changes in the commit; (v) performs the selection of the content considering the threshold values defined by the developer; and (vi) finally, it generates the change descriptions for for the analyzed commit. The tool proposed by Moreno *et al.* [80], called ARENA, extended the work by Cortes *et al.* [28], extracting the information about the change types from two different sources of information namely, the versioning system and the source archives of the releases to be com- pared. ARENA and Change Scribe achieved high accuracy in generating high quality commit messages and release notes. Indeed, in some cases, according to the involved study participants, they were often preferred to the one written by the original developers.

6.2.2 Summarization Techniques for Testing and Code Reuse

As discussed previously, most of previous work on source code and code change summarization have been evaluated by simply surveying developers about the general quality of the provided summaries [14, 117, 118, 74, 55, 79]. However, recent work proposed summarization techniques to support developers during bug fixing and/or testing tasks [64, 93], demonstrating their practical usefulness in performing such tasks.

Specifically, since *Waterfall* up until *Agile*, Software Testing has been playing an essential role in any software development methodology to detect defects of software products in different target environments. However, developers perceived testing as a time-consuming task because it requires a quarter of their working time engineering tests [13] and up to 50% of the overall project effort [15]. In this scenario, automated tests generation tools [46, 45] in software development pipelines could potentially reduce the time spent by developers in writing test cases. The main advantages of such tools include the generation of tests achieving higher code coverage when compared to the coverage obtained through manual testing [46] and to find violations of undeclared exceptions [45].

Despite such undisputed advantages, nowadays, automated test generation tools are still not used in practice. The main reason is that the generated tests are too hard to understand and difficult to maintain [29, 93]. As a consequence, generated tests do not improve the ability of developers to detect faults when compared to manual testing [46, 20]. Thus, to foster the adoption of automated testing tools, Panichella *et al.* [93] presented *TestDe-*

scriber, a tool that summarizes both automatically generated or manually-written JUnit tests cases. Specifically, taking as input the JUnit test and the corresponding class under test (CUT), TestDescriber (i) runs each test method tracing statements and branches exercised in the CUT (iii) augments the JUnit test with summaries that, at different abstraction levels, provides a dynamic view of the CUT.

Having TestDescriber to their disposal, Panichella *et al.* [93] performed an empirical study [93] to investigate the usefulness of the proposed tool when used in a concrete scenario of use: a Java class has been modified or developed and must be tested using generated test cases with the purpose of identifying and fixing eventual bugs affecting the production code. Thus, the goal of the study was to determine the impact of the generated test summaries on the number of bugs actually fixed by developers when assisted by automated test generation tools. Results of our study show that participants without TestDescriber summaries were able to remove only 40% of bugs present on the considered classes. Instead, when relying on test case with summaries, TestDescriber improved the bug fixing performance of the participants from 50% up to 100%. The results of the Wilcoxon test highlighted that the result was statistically significant (with p-values always < 0.05).

Thus, differently from most work in literature the work by Panichella *et al.* [93] is the first that deal with the defect or bug detection. Indeed, it is important to mention that other recent work in literature proposed the use of NLP templates and or summarization techniques mostly to document undocumented part of source code, without addressing the problem of bugs detection [85, 81]. Following, this line of research, Zhou *et al.*[130] proposed an approach based on NLP templates, able to detect *API defects* in Java Libraries. Specifically, Application Programming Interfaces (APIs) represent the most adopted tools for developers to build complex software systems nowadays. However, several studies have revealed that also major APIs providers tend to have an incomplete or inconsistent API documentation. This severely hampers the APIs comprehension and the quality of software built on it. Thus, Zhou *et al.*[130] proposed DRONE, a framework to automatically detect and repair defects affecting API documents by leveraging techniques from program analysis, natural language processing, and constraint solving. The research evaluation involving part of well documented JDK 1.8 APIs have shown that DRONE is able to detect API defects with an average F-measure of 79.9%, 71.7%, and 81.4%, respectively, demonstrating it usefulness.

6.3 Summarization of Textual User Feedback

In the current software industry developers are involved in a fierce competition to acquire and retain users. Thus, in this competitive market un-

derstanding the factors affecting users' experience and satisfaction and how these factors are related to software quality represents a valuable benefit for developers interested to evolve their software applications. In this context, app Stores, such as Google Play or the Apple Store, allow users to provide feedback on apps by posting review comments and giving star ratings. The experience an end-user has with apps reported in user reviews is a key concern when creating and maintaining any successful application. For this reason, mobile developers are interested to exploiting opinions and/or feedback of end-users during the evolution of their software [125, 24].

As discussed previously, automatically generated summaries can be either *indicative*, *informative*, or *critical* [54] Specifically, a ***critical summary (or review)*** reports or selects the main opinions or statements related to a specific discussed topic, thus, it brings the most relevant feedback, both positives and negatives, about a given subject discussed in the source document. Thus, the peculiarity of critical summaries has pushed researchers to conceive tools for automatically extracting user feedback from user review, relevant for software evolution [24, 52, 53, 62, 73, 90, 89, 27] For instance, Chen et al. presented a computational framework which automatically groups, prioritizes and visualizes informative reviews [24]. However, most of proposed tools only perform a simple classification of user reviews according to specific topics [53, 52, 87, 90, 62, 90], without reducing the amount of reviews developers have to deal with, which is very large for popular apps.

A more recent work By Ciurumelea *et al.*[27] proposed an approach that classify reviews according to more fine grained topics addressed by users in app reviews, which uses Machine Learning to classify reviews according to such topics. In addition, to dealing with such amount of user review data, Di Sorbo *et al.* [35, 36] proposed an approach called SURF, which at the same time, (i) determines the specific topic discussed in the review (e.g., UI improvements, security issues, etc.), (ii) identifies the maintenance task to perform for addressing the request stated in the review (e.g., bug fixing, feature enhancement, etc.), (iii) present such information to developers as an actionable *condensed, interactive and structured agenda of recommended software changes*. The approach relies on a conceptual model of the user requests reported by app reviews, then it uses sophisticated summarisation approaches, based on machine learning and NLP techniques, for summarizing thousands of reviews. Di Sorbo *et al.* [35, 36] performed an end-to-end evaluation of the proposed approach on user reviews of 17 mobile apps and involving 23 developers. Results demonstrates high accuracy of SURF in summarizing reviews containing feedback for planning future software changes, substantially reducing the the time and effort required for manually analyzing user review content.

6.4 Future Research

Recent research in SE observed an increasing adoption of summarization techniques for accomplishing simple or more complex, development, maintenance and testing tasks. However, their adoption in industrial contexts requires substantial novel and advanced research to make them applicable in any industrial or open source organizations. Thus, future research in the field is devote to fill the existing gap between industrial needs and current provided research prototypes:

- *Summarization of Hetherogeneous Data*: current summarization approaches are mostly conceived for analyzing one or two sources of information. However, when performing development, maintenance and testing tasks developers access to various types of heterogenous data. Thus, future Summarization techniques should be designed with advanced mechanisms able to distill, in a simultaneous manner, the relevant knowledge present in different sources of information, presenting it in a unified manner, depending on the specific task the developers is performing.
- *Scalability and integration in the CD/CI process*: current summarization approaches are able to proficiently distill relevant information from various kind of software artifacts. However, they are usually computationally expensive and thus, not applicable in real working contexts. Moreover, most of such tools are difficult to integrate in the current continuous delivery software development process. Hence, future research should be devoted on designing tools able to analyze, with substantial low computational cost, the huge amount of available heterogeneous data, integrating the summarized information in the the various development phases composing the CD pipeline applied in a software organization.
- *Visualization of Software Summaries*: most of generated software summaries are presented as set of textual fragments that share similar concepts. Thus, part of the future research related to the application of source code and code change summarization, needs to be devoted to the definition of proper visualization metaphors, that actually present the information provided by the generated summaries in a more structured manner.

7 Summary

Today's software development is about data and processes. Environments and tools are cornerstones for any successful project. At the same time we need to adopt new techniques from research to deal with this sheer amount of information that ranges from requirements to code to tests and to deployment and experimentation.

We discussed several techniques and their potential for these data-driven decisions and actions that are ready for use in practice. The techniques that we presented originate mostly from our own research and provide new solutions to recommender systems, automated test case generation, performance testing, continuous integration and continuous deployment, evolution analysis feedbacks, and summarization techniques for code or tests.

The next step is to adopt and integrate these and other techniques in the daily development activities and engineering processes. We showed some of the great potential of these technologies in dealing with the large amount of data that is available in any process step of software development.

New development environments are rising that support live development (in the cloud). Such tools require data that is beyond static code or test data; we need to build active feedback loops into the programming environments that employ proper data analytics customized for each development step. Tools such as live programming or programming in the cloud will then be eased and be put on a much more stable basis. With the increasing speed of software delivery to customers these links between the developer and the customer need to become seamless and active, building on the data aggregated, accumulated and analyzed for recommendations, summarizations, or testing in various dimensions.

References

1. Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/ SPEC International Conference on Performance Engineering (ICPE)*, pages 37–48, 2016.
2. Carol V. Alexandru, Sebastiano Panichella, and Harald Gall. Reducing redundancies in multi-revision code analysis. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Klagenfurt, Austria, 2017.
3. Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Softw. Eng.*, 36(6):742–762, November 2010.
4. M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A Survey of Machine Learning for Big Code and Naturalness. *ArXiv e-prints*, September 2017.
5. Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 281–293, New York, NY, USA, 2014. ACM.
6. Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM.
7. Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 207–216, Piscataway, NJ, USA, 2013. IEEE Press.

8. Sven Amann, Sebastian Proksch, and Sarah Nadi. FeedBaG: An Interaction Tracker for Visual Studio. In *International Conference on Program Comprehension*. IEEE, 2016.
9. Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. A Study of Visual Studio Usage in Practice. In *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016.
10. Alberto Bacchelli, Tommaso Dal Sasso, Marco D'Ambros, and Michele Lanza. Content classification of development emails. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 375–385, 2012.
11. Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.
12. Boris Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.
13. Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015. To appear.
14. Dave Binkley, Dawn Lawrie, Emily Hill, Janet Burge, Ian Harris, Regina Hebig, Oliver Keszocze, Karl Reed, and John Slankas. Task-driven software summarization. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 432–435. IEEE, 2013.
15. Frederick P. Jr. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.
16. Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.
17. Lubomír Bulej, Tomáš Bureš, Vojtěch Horký, Jaroslav Kotrč, Lukáš Marek, Tomáš Trojánek, and Petr Tůma. Unit testing performance with stochastic performance logic. *Automated Software Engineering*, 24(1):139–187, Mar 2017.
18. Raymond P.L. Buse and Westley R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 33–42. ACM, 2010.
19. Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. Syntax errors just aren't natural: Improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 252–261, New York, NY, USA, 2014. ACM.
20. Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D. Nguyen, and Paolo Tonella. Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. *ACM Trans. Softw. Eng. Methodol.*, 25(1):5:1–5:38, 2015.
21. Kai H Chang, JAMES H CROSS II, W Homer Carlisle, and Shih-Sung Liao. A performance evaluation of heuristics-based test case generation methods for software branch coverage. *International Journal of Software Engineering and Knowledge Engineering*, 6(04):585–608, 1996.
22. Lianping Chen. Continuous Delivery: Huge Benefits, but Challenges Too. *Software, IEEE*, 32(2):50–54, Mar 2015.
23. Lianping Chen. Continuous delivery: Overcoming adoption challenges. *Journal of Systems and Software*, 128:72–86, 2017.
24. Ning Chen, Jialiu Lin, Steven C. H. Hoi, Xiaokui Xiao, and Boshen Zhang. Arminer: Mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 767–778, New York, NY, USA, 2014. ACM.

25. Jürgen Cito, Philipp Leitner, Thomas Fritz, and Harald C. Gall. The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 393–403, New York, NY, USA, 2015. ACM.
26. Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An empirical analysis of the docker container ecosystem on github. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 323–333, Piscataway, NJ, USA, 2017. IEEE Press.
27. Adelina Ciurumelea, Andreas Schaufelbühl, Sebastiano Panichella, and Harald Gall. Analyzing reviews and code of mobile apps for better release planning. In *2017 IEEE 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 91–102, 2017.
28. Luis Fernando Cortes-Coy, Mario Linares Vásquez, Jairo Aponte, and Denys Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 275–284. IEEE, 2014.
29. Ermira Daka, Jose Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015. To appear.
30. Marco D'Ambros, Michele Lanza, and Romain Robbes. Commit 2.0. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, Web2SE '10, pages 14–19. ACM, 2010.
31. Kostadin Damevski, David Shepherd, Johannes Schneider, and Lori Pollock. Mining Sequences of Developer Interactions in Visual Studio for Usage Smells. *IEEE Transactions on Software Engineering*, 2016.
32. Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Using IR methods for labeling source code artifacts: Is it worthwhile? In *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, pages 193–202, 2012.
33. Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Labeling source code with information retrieval methods: an empirical study. *Empirical Software Engineering*, 19(5):1383–1420, 2014.
34. Augusto Born de Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. Perphecy: Performance regression test selection made simple but effective. In *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, Japan, 2017.
35. A. Di Sorbo, S. Panichella, C. Alexandru, J. Shimagaki, C.A. Visaggio, G. Canfora, and H.C. Gall. What would users change in my app? summarizing app reviews for recommending software changes. In *Foundations of Software Engineering (FSE), 2016 ACM SIGSOFT International Symposium on the*, pages 499–510, 2016.
36. Andrea Di Sorbo, Sebastiano Panichella, Carol V Alexandru, Corrado A Visaggio, and Gerardo Canfora. Surf: Summarizer of user reviews feedback. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 55–58. IEEE Press, 2017.
37. Martin Dias, Damien Cassou, and Stephane Ducasse. Representing Code History with Development Environment Events. In *International Workshop on Smalltalk Technologies*, 2013.
38. Robert Dyer. *Bringing Ultra-large-scale Software Repository Mining to the Masses with Boa*. PhD thesis, Ames, IA, USA, 2013. AAI3610634.
39. Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

40. Aleksander Fabijan, Pavel Dmitriev, Helena Holmström Olsson, and Jan Bosch. The evolution of continuous experimentation in software product development. In *International Conference on Software Engineering*, ICSE, Buenos Aires, 2017.
41. Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.
42. Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, November 2007.
43. Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM, 2011.
44. Gordon Fraser and Andrea Arcuri. Whole Test Suite Generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.
45. Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2015.
46. Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 291–301. ACM, 2013.
47. H.C. Gall, B. Fluri, and M. Pinzger. Change analysis with evolizer and changedistiller. *Software, IEEE*, 26(1):26–33, 2009.
48. Matthew J Gallagher and V Lakshmi Narasimhan. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, 23(8):473–484, 1997.
49. Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.
50. David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.
51. Giovanni Grano, Andrea Di Sorbo, Francesco Mercaldo, Corrado Aaron Visaggio, Gerardo Canfora, and Sebastiano Panichella. Android apps and user feedback: a dataset for software evolution and quality improvement. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics, WAMA@ESEC/SIGSOFT FSE 2017, Paderborn, Germany, September 5, 2017*, pages 8–11, 2017.
52. E. Guzman and W. Maalej. How do users like this feature? a fine grained sentiment analysis of app reviews. In *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*, pages 153–162, Aug 2014.
53. E. Ha and D. Wagner. Do android users write about electric sheep? examining consumer reviews in google play. In *Consumer Communications and Networking Conference (CCNC), 2013 IEEE*, pages 149–157, Jan 2013.
54. Udo Hahn and Inderjeet Mani. The challenges of automatic summarization. *Computer*, 33(11):29–36, November 2000.
55. Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 35–44. IEEE, 2010.
56. Maen Hammad, Anas Abuljadayel, and Mohammad Khalaf. Automatic summarising: The state of the art. *Lecture Notes on Software Engineering*, 4(2):129–132, 2016.

57. Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated Root Cause Isolation of Performance Regressions During Software Development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 27–38, New York, NY, USA, 2013. ACM.
58. Vincent J. Hellendoorn, Premkumar T. Devanbu, and Alberto Bacchelli. Will they like this?: Evaluating code contributions with language models. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 157–167, Piscataway, NJ, USA, 2015. IEEE Press.
59. Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.
60. Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance Regression Testing Target Prioritization via Performance Risk Analysis. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 60–71, New York, NY, USA, 2014. ACM.
61. Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.
62. Claudia Iacob and Rachel Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 41–44, Piscataway, NJ, USA, 2013. IEEE Press.
63. Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.
64. M. Kamimura and G.C. Murphy. Towards generating human-oriented summaries of unit test cases. In *Proc. of the International Conference on Program Comprehension (ICPC)*, pages 215–218. IEEE, May 2013.
65. Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. In *30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 41–50. IEEE, 2014.
66. Junhwi Kim, Byeonghyeon You, Minhyuk Kwon, Phil McMinn, and Shin Yoo. Evaluating CAVM: A new search-based test data generation tool for C. In *International Symposium on Search-Based Software Engineering (SSBSE 2017)*, 2017.
67. Andrew J Ko, Brad A Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 199–206. IEEE, 2004.
68. E. Kocaguneli, T. Menzies, and J.W. Keung. On the value of ensemble effort estimation. *Software Engineering, IEEE Transactions on*, 38(6):1403–1416, 2012.
69. Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. Online Controlled Experiments at Large Scale. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1168–1176, New York, NY, USA, 2013. ACM.
70. Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. *SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs*, pages 712–717. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
71. Kiran Lakhotia, Mark Harman, and Hamilton Gross. Austin: An open source tool for search based software testing of c programs. *Information and Software Technology*, 55(1):112–125, 2013.
72. Philipp Leitner and Cor-Paul Bezemer. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 373–384, New York, NY, USA, 2017. ACM.

73. W. Maalej and H. Nabil. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *Requirements Engineering Conference (RE), 2015 IEEE 23rd International*, pages 116–125, Aug 2015.
74. Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 279–290. ACM, 2014.
75. Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action: Covers Apache Lucene 3.0*. Manning Publications Co., 2010.
76. Phil McMinn. Search-based software testing: Past, present and future. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 153–163, Washington, DC, USA, 2011. IEEE Computer Society.
77. Thilo Mende and Rainer Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE '09, pages 7:1–7:10. ACM, 2009.
78. Roberto Minelli, Andrea Mocci, Romain Robbes, and Michele Lanza. Taming the ide with fine-grained interaction data. In *International Conference on Program Comprehension*, 2016.
79. L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 23–32. IEEE, May 2013.
80. Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 484–495, 2014.
81. Laura Moreno and Andrian Marcus. Automatic software summarization: the state of the art. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 511–512, 2017.
82. Evan Moritz, Mario Linares-Vásquez, Denys Poshyvanyk, Mark Grechanik, Collin McMillan, and Malcom Gethers. Export: Detecting and visualizing api usages in large source code repositories. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 646–651. IEEE Press, 2013.
83. Gail C. Murphy. *Lightweight Structural Summarization As an Aid to Software Evolution*. PhD thesis, 1996. AAI9704521.
84. G.C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Elipse IDE? *IEEE Software*, 2006.
85. Najam Nazar, Yan Hu, and He Jiang. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology*, 31(5):883–909, Sep 2016.
86. Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E Johnson, and Danny Dig. Is it dangerous to use version control histories to study source code evolution? In *European Conference on Object-Oriented Programming*, pages 79–103. Springer, 2012.
87. Fabio Palomba, Pasquale Salza, Adelina Ciurumelea, Sebastiano Panichella, Harald C. Gall, Filomena Ferrucci, and Andrea De Lucia. Recommending and localizing change requests for mobile apps based on user reviews. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 106–117, 2017.
88. Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *ICST*, pages 1–10. IEEE Computer Society, 2015.
89. S. Panichella, A. Di Sorbo, E. Guzman, C.A. Visaggio, G. Canfora, G. Gall, H.C., and H.C. Gall. Ardoc: App reviews development oriented classifier. In *Foundations*

of Software Engineering (FSE), 2016 ACM SIGSOFT International Symposium on the, pages 1023–1027, 2016.

90. S. Panichella, A. Di Sorbo, E. Guzman, C.A. Visaggio, G. Canfora, and H.C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 281–290, 2015.
91. Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo Canfora. Mining source code descriptions from developer communications. In *Proceedings of the International Conference on Program Comprehension, ICPC*, pages 63–72. IEEE, 2012.
92. Sebastiano Panichella, Gabriele Bavota, Massimiliano Di Penta, Gerardo Canfora, and Giuliano Antoniol. How developers' collaborations identified from different sources tell us about code changes. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 251–260, 2014.
93. Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 547–558, New York, NY, USA, 2016. ACM.
94. Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Prompter: A self-confident recommender system. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 577–580. IEEE, 2014.
95. Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Summarizing complex development artifacts by mining heterogeneous data. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 401–405, Piscataway, NJ, USA, 2015. IEEE Press.
96. Sebastian Proksch, Sven Amann, and Sarah Nadi. Enriched Event Streams: A General Dataset For Empirical Studies On In-IDE Activities Of Software Developers. In *International Conference on Mining Software Repositories (Accepted Mining Challenge)*, 2017.
97. Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. Evaluating the Evaluations of Code Recommender Systems: A Reality Check. In *International Conference on Automated Software Engineering*. ACM, 2016.
98. Sebastian Proksch, Veronika Bauer, and Gail C Murphy. How to Build a Recommendation System for Software Engineering. In *Software Engineering*. Springer, 2015.
99. Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent Code Completion with Bayesian Networks. *Transactions of Software Engineering and Methodology. ACM*, 2015.
100. Sebastian Proksch, Sarah Nadi, Sven Amann, and Mira Mezini. Enriching In-IDE Process Information with Fine-grained Source Code History. In *International Conference on Software Analysis, Evolution, and Reengineering*, 2017.
101. Stevche Radevski, Hideaki Hata, and Kenichi Matsumoto. Towards building api usage example metrics. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 619–623. IEEE, 2016.
102. Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. Summarizing software artifacts: A case study of bug reports. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 505–514, 2010.
103. Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. Automatic summarization of bug reports. *IEEE Trans. Software Eng.*, 40(4):366–380, 2014.
104. Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439, New York, NY, USA, 2016. ACM.

105. Martin P Robillard. What makes apis hard to learn? answers from developers. *IEEE software*, 26(6), 2009.
106. José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 338–349, New York, NY, USA, 2015. ACM.
107. Mohamed Aymen Saied, Omar Benomar, Hani Abdeen, and Houari Sahraoui. Mining multi-level api usage patterns. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 23–32. IEEE, 2015.
108. Mohamed Aymen Saied, Omar Benomar, Hani Abdeen, and Houari Sahraoui. Mining multi-level api usage patterns. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 23–32. IEEE, 2015.
109. Simone Scalabrino, Giovanni Grano, Dario Di Nucci, Rocco Oliveto, and Andrea De Lucia. Search-Based Testing of Procedural Programs: Iterative Single-Target or Multi-target Approach? In *Search Based Software Engineering*, pages 64–79. Springer, Cham, Cham, October 2016.
110. Gerald Schermann, Jürgen Cito, Philipp Leitner, and Harald C Gall. Towards Quality Gates in Continuous Delivery and Deployment. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4. IEEE, 2016.
111. Gerald Schermann, Jürgen Cito, Philipp Leitner, Uwe Zdun, and Harald C. Gall. We're Doing It Live: An Empirical Study on Continuous Experimentation. *Journal of Information and Software Technology*, 2017. Under submission.
112. Gerald Schermann, Dominik Schöni, Philipp Leitner, and Harald C. Gall. Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies. In *Proceedings of the 17th International Middleware Conference*, pages 12:1–12:14, New York, NY, USA, 2016. ACM.
113. Andrea Di Sorbo, Sebastiano Panichella, Corrado A Visaggio, Massimiliano Di Penta, Gerardo Canfora, and Harald C Gall. Development emails content analyzer: Intention mining in developer discussions. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 12–23. IEEE, 2015.
114. Karen Spärck Jones. Automatic summarising: The state of the art. *Inf. Process. Manage.*, 43(6):1449–1481, 2007.
115. G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 101–110. IEEE, 2011.
116. G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Proceedigs of the International Conference on Program Comprehension (ICPC)*, pages 71–80. IEEE, 2011.
117. Giriprasad Sridhara. *Automatic Generation of Descriptive Summary Comments for Methods in Object-oriented Programs.* PhD thesis, Newark, DE, USA, 2012. AAI3499878.
118. Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 43–52. ACM, 2010.
119. Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr Tuma. Unit Testing Performance in Java Projects: Are We There Yet? In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 401–412, New York, NY, USA, 2017. ACM.
120. Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 269–280, New York, NY, USA, 2014. ACM.

121. Michael VanHilst, Shihong Huang, James Mulcahy, Wayne Ballantyne, Ed Suarez-Rivero, and Douglas Harwood. Measuring effort in a corporate repository. In *IRI*, pages 246–252. IEEE Systems, Man, and Cybernetics Society, 2011.
122. Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, and Gerardo Canfora. Codes: Mining source code descriptions from developers discussions. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 106–109. ACM, 2014.
123. Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A tale of ci build failures: an open source and a financial organization perspective. 2017.
124. Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. Continuous delivery practices in a large financial organization. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, pages 519–528, 2016.
125. Tejas Vithani. Modeling the mobile application development lifecycle. In *Proceedings of the International MultiConference of Engineers and Computer Scientists 2014, Vol. I*, IMECS 2014, pages 596–600, 2014.
126. E. Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 562–567. IEEE, 2013.
127. Ho Chung Wu, Robert Wing Pong Luk, Kam Fai Wong, and Kui Lam Kwok. Interpreting tf-idf term weights as making relevance decisions. *ACM Transactions on Information Systems (TOIS)*, 26(3):13, 2008.
128. Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57. ACM, 2006.
129. Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. Example overflow: Using social media for code recommendation. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, pages 38–42. IEEE Press, 2012.
130. Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald C. Gall. Analyzing apis documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 27–37, 2017.